MUSE: MTD in Userspace

2023-28-06

Richard Weinberger

Hello

Richard Weinberger

- Co-founder of sigma star gmbh
- Linux kernel developer and maintainer
- Strong focus on Linux kernel, lowlevel components, virtualization, security, code audits

sigma star gmbh

- Software Development & Security Consulting
- Main areas: Embedded Systems, Linux Kernel & Security
- Contributions to Linux Kernel and other OSS projects

Agenda

- Motivation: Why MUSE?
- MUSE implementation: Why FUSE?
- MUSE details
- FUSE internals: How you can (ab)use it

Motivation

- Testing components ontop of MTD can be unpleasant
- For me interesting components are UBI, UBIFS and JFFS2
- What we have so far:
  - In-kernel: mtdram, block2mtd, nandsim
  - Using virt: qemu, etc.

In-kernel: mtdram

- Operates on a `vmalloc()`'ed memory region
- Implememts MTD interface (not a MTD subsystem such as NAND, NOR, SPI-NOR)
- Type `MTD_RAM`
- Useful to simulate small MTDs such as parallel NOR chips
- Allows only one instance
- Good enough for basic JFFS2 testing
- No fault nor error injection support

In-kernel: block2mtd

- Operates on a given block device
- Just like mtdram, implememts a MTD interface
- Type `MTD_RAM`
- Allows more instances
- Good enough for basic UBI and UBIFS testing (NOR mode)
  - No wear leveling
- No fault nor error injection support

In-kernel: nandsim

- Operates on `kmalloc()`'ed NAND pages
- Can swap pages to a file
- Mocks a parallel NAND *chip*
- Implements MTD NAND framework
- Good for UBI and UBIFS testing
- Support for: ECC, parts, error injection, delays
- Slow and error prone
- NAND geometry via NAND IDs
- Goal was finding errors in MTD NAND and UBI subsystems
  - These days we find mostly bugs in nandsim itself ;-)

Virt: QEMU (or any other)

- Can emulate flash devices
- Mostly for UEFI guest support
- For my use case too inflexible
- No fault nor error injection support

My wishlist

- Add and remove MTD at runtime
- Support NOR and NAND style MTDs
- Support for various image formats
  - With and without OOB
  - Vendor specific
- User controllable error and fault injection support

Idea

- Create a new MTD simulator
- Simulate NAND, NOR, SPI-NAND, SPI-NOR, etc..
- Keep kernel component simple and stupid
- Do all hard work in userspace

First try: ad-hoc

- Create a new interface to control a MTD simulator from userspace
- Reinventing the wheel

Second try: qemu/virtio

- Have a generic MTD in qemu
- Plus a generic MTD driver in kernel
- Let userspace (virt host) control the device
- Didn’t really fit my needs

Third try: FUSE/CUSE

- Co-worker: “Can’t you mock MTD characteristics in userspace using CUSE?”
- CUSE: Character device in userspace: special operation mode of FUSE
- When you have read, write, ioctl, ..., you can do a character device
- A bare character device is nice but still no real MTD
- Kernel MTD subsystem will not know it
- But I liked the idea

Third try: FUSE/CUSE (cont'd)

- FUSE: Filesystem in userspace
- Filesystem ops (read, write, ioctl, stat, ...) implemented in userspace
- Rather generic
- Many users: e.g: sshfs, ntfs-3g
- Enough to implement an MTD
- MTD has no zero copy and other fancy IO: makes things easy

MUSE: MTD in userspace

- Add new FUSE operations to make MTD happy
- `MUSE_READ`, `MUSE_WRITE`, `MUSE_ERASE`, `MUSE_ISBAD`, `MUSE_MARKBAD`, `MUSE_SYNC`
- OOB, ECC support
- MTD lifetime was hard to get right

MUSE: Features (in progress)

- Snapshots
- Custom image types (not just nanddump)
- Record/replay
- Fault injection
- Fuzzing

MUSE: Status

- Kernel part almost done, less than 1000 LoC
- Still experimenting with userspace
  - Playing with Rust

MUSE for non-testing

- Real MTD drivers are possible too
- Only if flash device is fully controllable via userspace
  - Hint: spidev and UIO help
- I *do not* recommend this except for PoC drivers

More on FUSE

- Server/client architecture
- Userspace is the server!
- Kernel side implements a generic driver
  - VFS in case of FUSE
  - miscdevice in case of CUSE
  - MTD for MUSE
  - ... your own

More on FUSE (cont'd)

- Communication is request based
- Requests are made by the kernel
- Each request contains an operation
- Userspace reacts on it
- Reply contains a per-operation reply structure

More on FUSE (cont'd)

- Usually each operation has an in and out sturcture
- Example: `FUSE_WRITE`
- `struct fuse_write_in` and `struct fuse_write_out`

```
struct fuse_write_in {
        uint64_t        fh;
        uint64_t        offset;
        uint32_t        size;
        uint32_t        write_flags;
        uint64_t        lock_owner;
        uint32_t        flags;
        uint32_t        padding;
};
```

```
struct fuse_write_out {
        uint32_t        size;
        uint32_t        padding;
};
```

More on FUSE (cont'd)

- An answer to a request contains most of the time three io vectors:
  1. `struct fuse_out_header`: Overall return code
  2. Operation specific out message, e.g: `struct fuse_write_out`
  3. Payload, a buffer with a length
- A request itself can also contain a buffer (think of write requests)

How to create your own userspace driver framework

1. Define new FUSE operations plus in/out structures
   - Ideally re-use existing ones!
   - They are UAPI!
   - `include/uapi/linux/fuse.h`
2. Implement a control character device (like `/dev/fuse`)
   - Userspace will use it to install new devices
   - In `open()` kernel will send `INIT` op
3. Implement a generic device driver
   - All interesting operations will create a request and use the result
4. Add your operations to libfuse_lowevel (or handle requests directly)

Example: MUSE_ISBAD

- Used by the kernel to test whether a block is bad
- Only userspace can know, so a request is needed

```
struct muse_isbad_in {
    uint64_t    addr;
};
```

```
struct muse_isbad_out {
    uint32_t    result;
    uint32_t    padding;
};
```

Example: MUSE_ISBAD (cont'd)

- Kernel side of the generic MTD driver

```
static int muse_mtd_isbad(struct mtd_info *mtd, loff_t addr)
{
[...]

    inarg.addr = addr;

    args.opcode = MUSE_ISBAD;
    args.nodeid = FUSE_ROOT_ID;
    args.in_numargs = 1;
    args.in_args[0].size = sizeof(inarg);
    args.in_args[0].value = &inarg;
    args.out_numargs = 1;
    args.out_args[0].size = sizeof(outarg);
    args.out_args[0].value = &outarg;

    ret = fuse_simple_request(fm, &args);
[..]
}
```

Example: MUSE_ISBAD (cont'd)

- libfuse_lowlevel side:

```
void do_muse_isbad(fuse_req_t req, fuse_ino_t nodeid, const void *inarg)
{
    struct muse_isbad_in *arg = (struct muse_isbad_in *)inarg;

    (void)nodeid;

    if (req->se->op.muse_block_isbad)
        req->se->op.muse_block_isbad(req, arg->addr);
    else
        fuse_reply_err(req, ENOSYS);
}
```

Example: MUSE_ISBAD (cont'd)

- Application side:

```
void my_mtd_isbad(fuse_req_t req, loff_t addr)
{
    int isbad = rand() & 1;

    muse_send_block_isbad_reply(req, 0, isbad);
}
```

Example: MUSE_ISBAD (cont'd)

› libfuse_lowlevel side:

```
int muse_send_block_isbad_reply(fuse_req_t req, int error, int isbad)
{
    struct iovec iov[2];
    struct muse_isbad_out out =  {
        .result = isbad,
    };
    int ret;

    iov[1].iov_base = (void *)&out;
    iov[1].iov_len = sizeof(out);

    ret = fuse_send_reply_iov_nofree(req, error, iov, 2);
    fuse_free_req(req);

    return ret;
}
```

Summary

- FUSE offers a nice and powerful framework
- You can do much more than filesystems in userspace
- Non-complex devices can be emulated with reasonable effort
- libfuse (and libfuse_lowlevel) offer most building blocks
    - Many helpers to create and process requests
    - Many examples and hints
- First MUSE PoC was ready within a day

FIN

Thank you!

Questions, Comments?

Richard Weinberger
richard@sigma-star.at